Measuring HT-Enabled Multi-Core: Advantages of a Thread-Oriented Approach

Sergey N. Zheltov
Project Manager and Senior Software Engineer
Software and Solutions Group
Intel Corporation

Stanislav V. Bratanov
Software Engineer
Software and Solutions Group
Intel Corporation

Table of Contents

(Click on page number to jump to sections)

MEASURING HT-ENABLED MULTI-CORE: ADVANTAGES OF A THREAD-ORIENTED APPROACH. 3

OVERVIEW: HANDLING MULTITHREADED CODE 3
KNOWN PERFORMANCE MONITORING METHODS 3
MONITORING SMP SYSTEMS 5
MONITORING SYMMETRIC MULTITHREADING SYSTEMS 7
SUMMARY 9
MORE INFO 10
AUTHOR BIOS 10

Measuring HT-Enabled Multi-Core: Advantages of a Thread-Oriented Approach

Sergey N. Zheltov
Project Manager and Senior Software Engineer
Software and Solutions Group
Intel Corporation

Stanislav V. Bratanov
Software Engineer
Software and Solutions Group
Intel Corporation

Overview: Handling Multithreaded Code

From a hardware point of view, multi-core processors are simply single-die physical packages that provide capabilities similar to traditional symmetric multiprocessing (SMP) machines, and inevitably inherit most of the performance monitoring problems a programmer typically faces while working with SMP.

In this article, we will address such performance estimation problems and advocate a thread-oriented approach to performance monitoring as an efficient way of dealing with multithreaded code. At the same time, we will also address this as a basis for solving the particular task of monitoring Hyper-Threading Technology (HT Technology)-enabled multi-core systems.

This article discusses several methods and provides solutions to the problems which cannot be solved without the notion of the actual thread layout over time, thread activity, and interaction.

The problem of monitoring the utilization of resources shared between multiple cores or CPUs is discussed in an example of measuring bus utilization. A solution based on per-thread monitoring is suggested, and compared with sampling-based approaches.

Also, a method to avoid some current limitations of performance monitoring in HT Technology-enabled processors and a method to increase the precision of measurements for Hyper-Threading Technology are discussed in detail, with the need for and advantages of per-thread monitoring also summarized.

Known Performance Monitoring Methods

Basic problems of performance monitoring are the same for both single- and multi-threaded applications since all modern operating systems provide multitasking execution models. The methods and issues discussed here are fully applicable to all multitasked execution environments and may be viewed with no regard to how many processors a particular system supports.

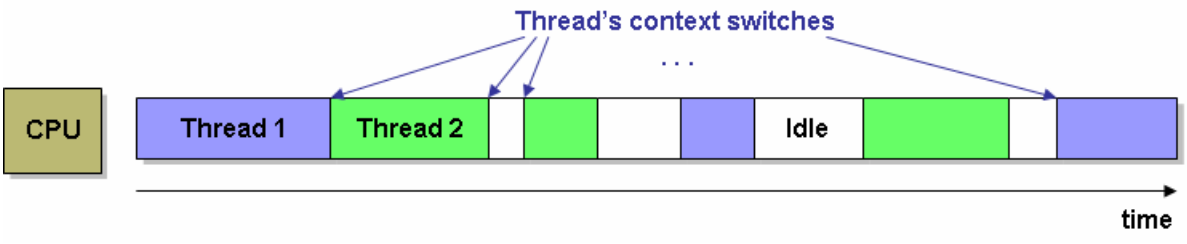

Figure 1. *Thread execution over time.*

Normally, the process of execution in a multitasking operating system flows as depicted in **Figure 1**. Threads running on a processor are eventually interrupted and suspended; other threads are selected by internal means and scheduled for execution, and so on. This imposes a performance monitoring rule for multithreaded environments: to measure the number of events (such as processor clock ticks) that have elapsed, the monitoring system needs to ensure that only

those events pertaining to a particular thread are to be counted for the particular CPU it is currently running. Otherwise, the results will depend on the number of processes and threads being currently executed and the state of the system which is generally not predictable.

There are several ways to achieve compliance with the above statement.

Direct Measurement

If the execution time of some code is long enough and the influence on execution of this code from other threads or the operating system is small enough (for example, user code execution time is much longer than the time spent in other threads or inside the OS, or measurement was done a short time before the context was switched) then the performance results obtained will likely be correct—influence from other threads or the OS can be considered as minimal (for the first case) or 0 (the second case). As this method implies too many assumptions and limitations, it cannot be viewed as reliable for multithreaded environment.

Sampling

The second way of filtering out the other thread events is to use a time- or event-based method of sampling (**Figure 2**) to check after a constant at a varying or adjustable time interval in which the thread (process) is currently active. As a result, performance monitoring data of the entire system is collected, and the further analysis may show how much time or how many events were spent on each thread execution.

This method is robust enough, though its precision inversely relates to the length of the sampling interval. At the same time, short intervals that are too short not only will slow down the thread execution, but will also cause frequent pipeline flushes and other changes to the processor's internal state. This in turn will lead to performance results that may not be considered representative samples and results that also may not be correct.

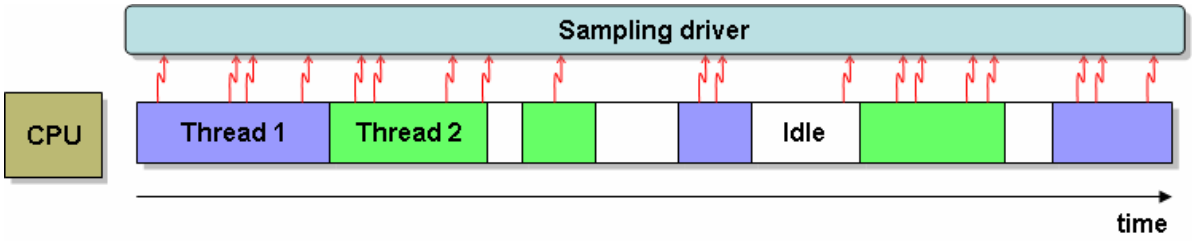

Figure 2. *Sampling process example.*

Handling Context Switches

The third method requires direct thread execution control, or at least context switch notification (**Figure 3**). Once a thread is switched for execution or suspended, the performance monitoring system may check whether this particular thread needs to be monitored and adjust the results.

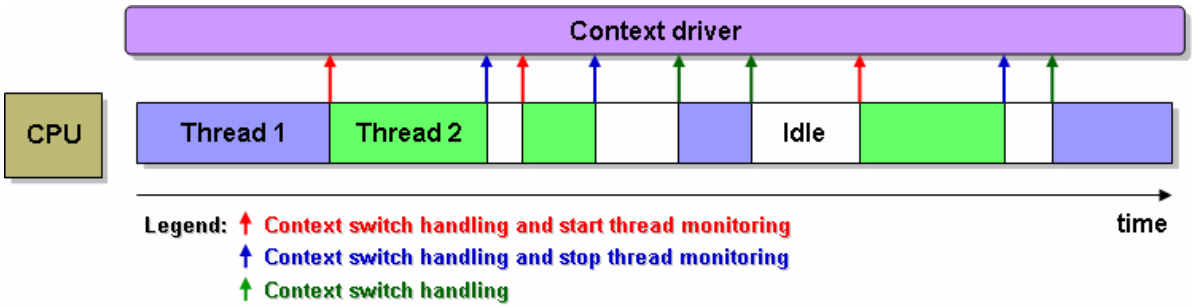

Figure 3. *Thread switch handling.*

As compared with the two previous methods, this method provides higher precision, which depends mostly on the context switch notification mechanism. It does not affect thread execution significantly because all internal computations are performed by the monitoring system on the "natural" borders of context switches. This means that changes to the processor's state would have appeared anyway.

Interesting results may be obtained from the combination of the sampling and thread switch control methods: each thread may be monitored independently, the execution time may be easily computed, and the event-based sampling performed within the thread activity timeframe may yield the real event-to-code distribution as shown in **Figure 4**.

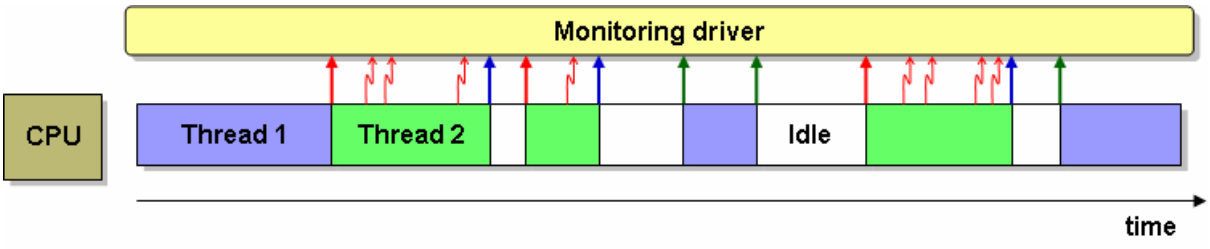

Figure 4. *Sampling on a thread basis.*

To complete the picture, it is worth noting that all performance monitoring results are usually combined into performance records (containing times, event amounts, and addresses), and these are saved for the off-line analysis or processed in real time and presented to a user as final values.

Monitoring SMP Systems

In order to monitor an SMP system (without regard to the particular method chosen), the monitoring software must establish control over each CPU to avoid undercounting when a thread or process is scheduled for execution on a processor that is not currently monitored. The abovementioned control requires either a monitoring (or sampling) procedure to run permanently on each processor, or, if a thread context switch is handled, switching to the destination processor when a particular thread of interest is activated.

Once the CPU control is established, more limitations may appear when it is necessary to monitor an event that indicates the usage of shared hardware resources. For example, if one needs to measure the utilization of the processor-to-memory controller bus, independent measurements of each processor's front side bus (FSB) activity are not sufficient.

Common Resource Utilization Problem

In this paragraph we'll consider a simplified example of memory bus utilization as example of common resource utilization between processors. The main assumptions here are uniform memory access distribution over time for each processor and also uniform distribution of memory access over processors in a multiprocessor system. Consider the following formula:

$$B = k\frac{T_{bdr}}{T}$$

Where ***B*** stands for bus utilization; ***k*** is the processor-to-bus frequencies ratio; T_{bdr} is the time (the number of clock ticks) the BUS_DATA_READY signal is active for the processor's front side bus. That is, when the data is transmitted, ***T*** is the time interval to average the utilization.

This formula describes bus utilization for a single-processor case, but in the case of an SMP system, this formula is computed for each processor and yields only the front side bus utilization for each processor. No information is contained on how much memory bandwidth (or external cache) the controller bus is utilizing in the entire system.

The formula to retrieve such information is the following:

$$B_{seq} = \frac{\sum_{n=0}^{N-1} B_n}{N}; \text{ and } B_{max} = \sum_{n=0}^{N-1} B_n;$$

where B_n is the average bus utilization computed for n-th processor; N is the number of processors in the system; B_{seq} corresponds to the average bus utilization for the entire system when the bus is accessed by all processors sequentially; and B_{par} denotes the average bus utilization for the parallel access case.

Parallel access implies that all processors have accessed the bus within the same time interval T that is used to compute the average utilization.

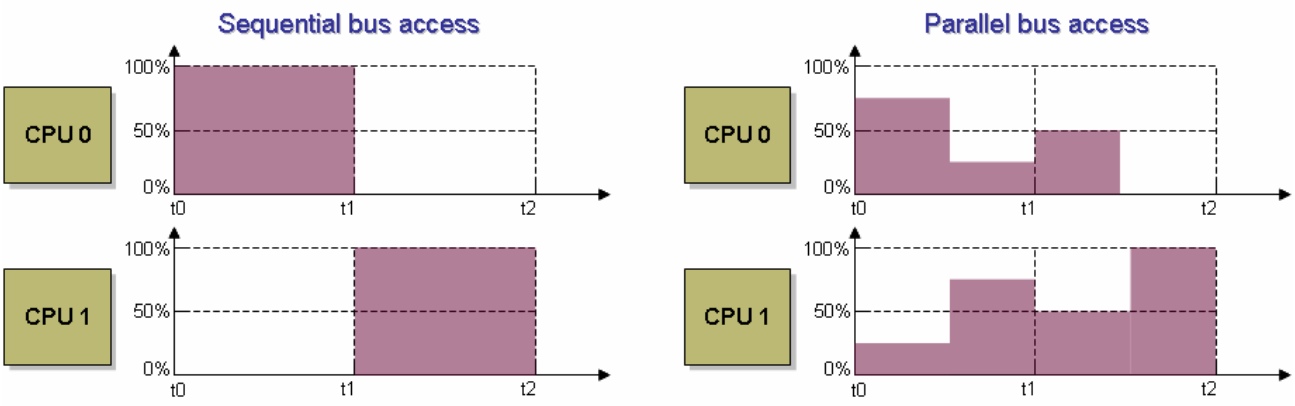

Figure 5. *Bus utilization graphs.*

Thus, according to **Figure 5**, the bus utilization B_n is averaged over equal time intervals `[T0, T1]` and `[T1, T2]`. The bus access is considered sequential if there is only one processor active during the time interval used to average the bus utilization. Otherwise, if there are more active processors within the averaging interval, the access is parallel.

In the example of Figure 5, we may calculate the bus utilization in the following ways:

- For the sequential case, the calculation comprises a sum of bus utilization during both intervals divided by the number of processors ((100% + 100%) / 2 = 100%).
- For the parallel case, bus utilization within each interval should be totaled (25% + 75%) and (50% + 50%), which yields 100% bus utilization.

Determining Parallel Regions

The above considerations show that, in order to obtain the value of bus bandwidth utilized by a multithreaded application in a multiprocessor system, the monitoring software should be able to determine which parts (threads) of the application are executed simultaneously by different processors, and which threads are run sequentially, delayed, or suspended.

Sampling-based systems, which bind event sampling to external time scale, are good enough to produce average results for rough analysis of complex applications that take noticeable time (up to several minutes) to execute. But the precision issues discussed in the previous section still apply to the SMP case, and even worse—the precision may degrade because it is impossible to determine exactly when parallel execution of thread 0 and thread 1 (in the example of **Figure 6**) begin if such a determination is based on asynchronous sampling interrupts.

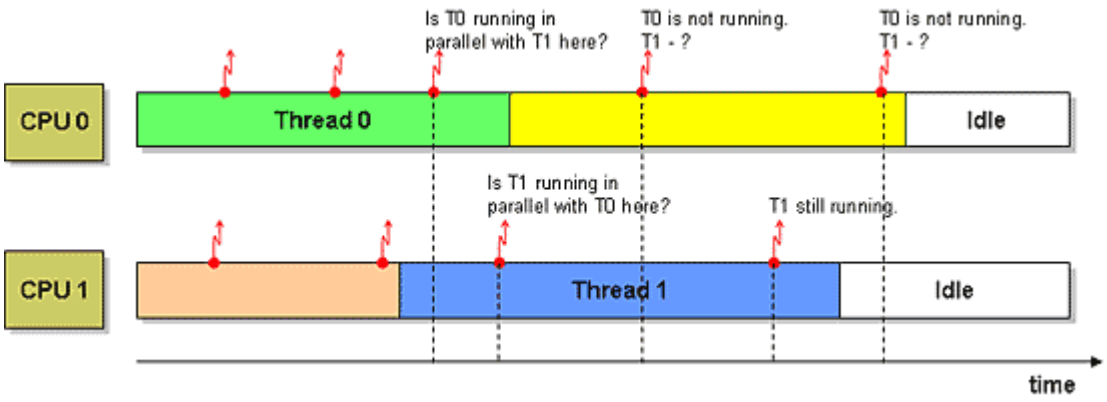

Figure 6. *Sampling-based determination of parallel execution regions.*

As shown in **Figure 7**, context switch information is absolutely necessary for correct computation of shared resource usage in SMP systems; moreover, the ability to "follow" the thread switch from processor to processor and to acquire information in context switch time and destination processor ID helps determine parallel execution for correct computations.

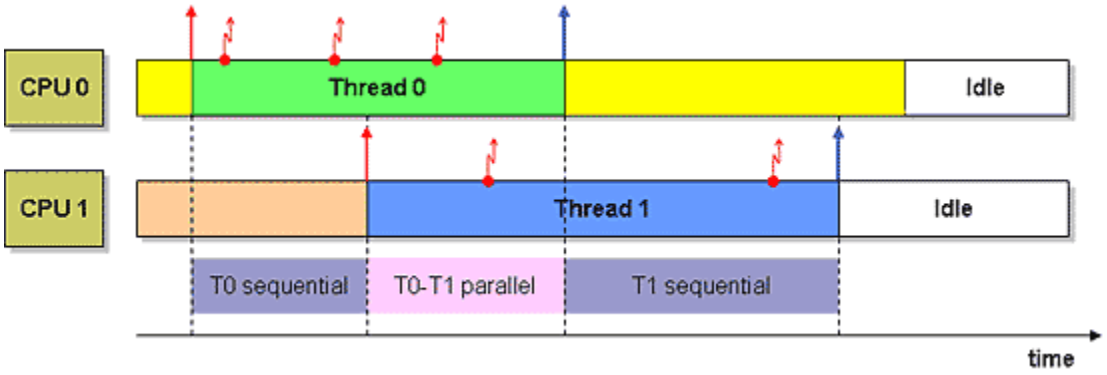

Figure 7. *Determining parallel execution regions based on thread-switch handling.*

Monitoring Symmetric Multithreading Systems

Symmetric Multithreading (SMT) systems do not differ much from SMP as seen by software unless they have internal resources that need to be shared by logical processors within one physical processor package. In this section, we will talk about Intel® architecture SMT systems based on HT Technology.

In current implementations of HT Technology, there are several performance events that can be measured for either the whole package or a subset of logical processors with one limitation: the subset selection is effective for all logical processors, that is, when one logical processor changes the subset, this setting affects performance counting of the others.

Dealing with Limitations

The sampling-based methods don't provide an accurate way to determine which threads are executed in parallel. SMT performance monitoring software implementations suffer from severe over-counting (if each logical processor is sampled independently) or are applicable only to physical packages, which are hardly acceptable for SMT.

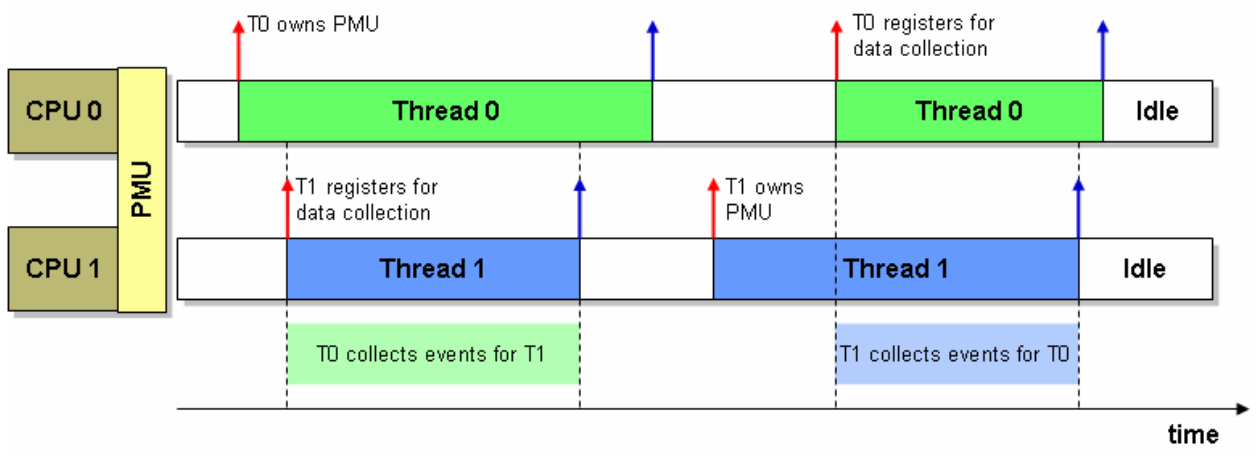

Figure 8. *SMT performance monitoring.*

However, if notified about a thread switch, the monitoring system may efficiently control shared physical resources between logical processors. One of the possible variants is illustrated in **Figure 8**. The first thread that is scheduled for execution in one of the logical processors takes ownership over its performance monitoring resource and starts counting. Other threads when scheduled to the same physical package need to check whether they are granted access to the resource. If not, they may add their identifiers to a wait list or to the subset of selected logical processors (see above) and continue execution. When the thread that owns the resource is suspended or switched to another processor, the monitoring software selects another thread from the wait list and grants resource ownership to that thread.

This allows all events to be counted correctly, but particular event-to-thread distributions remain distribution-model specific.

To prove the usefulness of the suggested SMT monitoring method, it is desirable to estimate its precision and compare it with other methods. The comparison is illustrated in **Figure 9**.

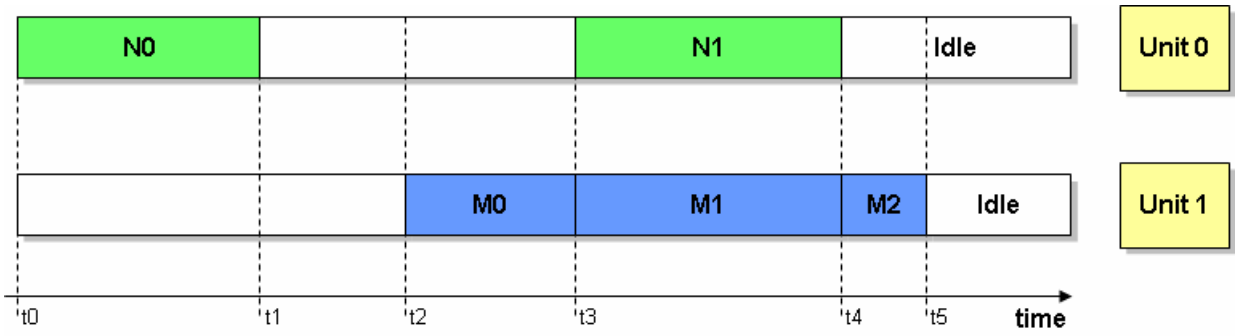

Figure 9. *Exemplary SMT resource usage over time.*

According to a non-SMT enabled method, the first unit to request the operation, or a predefined unit, (for instance, unit 0) collects all performance data pertaining to all units. In this case, all performance monitoring events that occurred within the time interval `[t0, t5]` would be attributed to unit 0. Suppose unit 0 actually performed `N=N0+N1` events, and unit 1 collected the value of `M=M0+M1+M2` events. The sum of N and M would be assigned to unit 0 (or unit 1). This may result in any precision ranging from 0 to 100 percent depending on the actual values of N and M (if N→0, then the precision also tends to zero, because the sum was assigned to an almost inactive unit).

According to the method suggested above, the last unit to request the performance monitoring operation indicates its active state to the unit that already collects performance data, and once the collecting unit becomes inactive, it signals the other unit to continue the performance data collection. For example, the number of events that occurred during the `[t0, t1]` time interval (let it be N0) are assigned to unit 0, and the sum of (M and N1) events that occurred within the time interval `[t2, t5]` are attributed to unit 1. The precision of such an approach also may range from 0 to 100 percent similarly to the first method, though this method is guaranteed against over-counting.

Raising the Precision

Better results may be achieved if the number of events within the `[t3, t4]` interval are distributed between the units as an average value or a weighted product of the total number of events within the interval and the probability assigned to each unit. This requires either the unit activity information be stored for further analysis or the following adaptive performance monitoring method be applied.

The essence of the adaptive method is to program a Performance Monitoring Unit (PMU) to collect performance information for the selected unit (in addition to other units that may have requested this operation earlier) and to generate a signal (an interruption) to the processor when any specified number of events is collected (much like the event-based sampling). The number of events may be predefined or adjustable in order to adapt the monitoring system to different activity intervals. For some cases of thread interaction, the less the number is, the more close the performance data is to the average; however, too small numbers may increase the overhead caused by frequent interruptions.

When the specified number of events elapses, the execution unit that owns the PMU is asynchronously interrupted and forced to perform an operation as follows: The current value of PMU may be read and assigned to the currently active execution unit (EU). Then a check for pending requests may be performed. In case there are no requests pending, the PMU may again be programmed to count the predefined or pre-estimated number of events for the current EU.

If there are other requests to be serviced, the selection of a new request (and corresponding EU) is carried out, and the PMU is programmed for another EU in a similar manner.

To better understand how this method affects the precision of performance monitoring, refer back to Figure 9. Here, the events that occurred within the time interval `[t0, t1]` are assigned to unit 0, the events of `[t2, t3]` and `[t4, t5]` intervals are attributed to unit 1; and the events collected during the time interval `[t3, t4]` are distributed in equal parts (or accordingly with the used model) between all EUs.

Thus, a system based on this method may yield the precision of measurements not less than 50 percent (in case of equally weighted execution units), dynamically adapt to changing activity intervals of EUs, and implement quasi-independent performance monitoring on a per-thread basis.

Applicability

This method of resource sharing may become very efficient when the program (or execution thread/unit) being monitored signals or specifies the type of its activity to be monitored, and thus, the monitoring system may decide whether particular processor resources need to be shared. That is, whether the processor's hardware needs to be reprogrammed to count another thread (execution unit) event upon receiving the next overflow signal.

The suggested method may also be applicable to nonsymmetric systems in case execution units (or processors) in such systems have the ability of receiving signals from PMUs and at least one shared resource fast enough to service performance data interchange.

Summary

As shown in this article, there are cases when correct results cannot be obtained without the notion of the actual thread activity over time, determination of parallel execution regions, and sharing performance monitoring resources on a per-thread basis.

The need exists for the capability to track execution context switches and design performance monitoring systems in such a manner that enables performance data collection for each execution thread independently.

An appropriately designed software monitoring system may lift the imposed restrictions and reduce the complexity of performance monitoring hardware. For example, if it is not technically possible to equip each logical processor within an SMP/SMT system with a dedicated performance monitoring unit, adequate results may be retrieved by employing the method of per-thread performance monitoring resource sharing described in this article.

To estimate the load of a shared resource within an SMP system, it is not necessary to attach a monitoring unit to the resource itself (to bus, memory, I/O controllers); it is enough to let each agent accessing the resource count its own activity when performed on a thread basis. This lets the monitoring system find points of resource (in time or code domain), over- or under-utilization, and at the same time, attribute the measured data to a particular thread of interest.

To summarize the advantages of having the actual information on thread activity over time vs. overall information on system performance:

- The overall information reflects the actual data that an end user will receive in response to his requests to an application (part of the entire computational system).
- Such overall information may vary significantly and even unpredictably based on particular conditions of the application's launch.
- There is a need for precise application-related information that will allow a developer to differentiate between system-induced events and an incorrect or nonoptimal application's behavior in order to answer the question whether the application can be improved and what the methods of improvement may be.

More Info

You can learn much more at the Intel Web site:

Intel® Software Network
Intel Multi-Core Processing site for software developers
Intel Multi-Core
The Evolution of Parallel Computing

References

1. Intel® Pentium® Processor Extreme Edition Technical Documents.
2. James Reinders. VTune™ Performance Analyzer Essentials.
3. William Stallings. Operating Systems: Internals and Design Principles (4th Edition).
4. oprofile.com: All the best resources on the net*.

Author Bios

Sergey Zheltov is a project manager and senior software engineer in the Advanced Computer Center (Software and Solutions Group). His research interests include parallel software and platforms architecture, operating systems, media compression and processing, signal processing, high-order spectra. He received a diploma in radio-physical engineering and M.S. degree in theoretical and mathematical physics from Nizhny Novgorod State University, Russia.

Stanislav Bratanov is a senior software engineer in the Software and Solutions Group. His research interests include multiprocessor software platforms, operating system kernels and environments, monitoring instruments, and platform-dependent media data coding. He graduated from Nizhniy Novgorod State University, Russia.

—End of Technology@Intel Magazine Article—